




C++ Fundamentals

WEDNESDAY, APRIL 2

Introduction to Fundamentals of C++ Programming

-  Readings: [Text](#) Ch. 1, 2.1-2.4
-  [Lecture quiz on Canvas](#)
-  [Lecture video on Canvas](#)

Lecture Video

Click to sign in and play video

Contents

0. Preliminary Note
1. Announcements
2. Productivity Tip: Hotkeys
3. Preliminaries: Syntax vs. Semantics
4. Preliminaries: Compilation and Execution
5. Getting Started: The Blank Qt Project (and the return of "Hello, world!")
6. Two Types of Comments in C++
7. **Glance over this!** A Brief Guide to Commenting
8. `#include` Directives
9. Namespaces
10. The `main()` Function
11. The `main()` Function is Special
12. Return Statements
13. The `cout` Stream
14. The `endl` String Manipulator

15. Variables and Data Types
16. Common Pitfalls with Variables
17. Uninitialized Variables
18. While Loops
19. The `++` Operator
20. Wonky Spacing
21. For Loops
22. For-Each Loops (Range-Based Loops)
23. Conditional (If-Else) Statements and Comparison Operators
24. Boolean Operators
25. Common Pitfall with the `||` Operator
26. **Catch-up Topic** Void Functions
27. **Catch-up Topic** Passing Parameters and Returning Values from Functions
28. **Catch-up Topic** Function Placement and Functional Prototypes
29. **Catch-up Topic** Variable Scope
30. What's next?
31. Exam Prep

Preliminary Note

Sections with the **Catch-up Topic** label in today's notes are topics that I covered in this lecture in a previous quarter, but didn't have time to get to in class today. Womp womp. :(We'll cover those topics on Friday (in addition to a **lot** of other material), but we'll move through them rather briskly. Please consider reading those ahead of time. I'll leave them on this page for now, until we see how our topic coverage shakes out over the next few days.

Announcements

Here are some announcements that are relevant to your life this week:

- Be sure to check out Jonathan's [Week #1 Announcements post](#) on Ed!
- Upcoming deadlines:
 - A0 due Friday 11:59 PM
 - Syllabus Quiz due Friday 11:59 PM
 - Section sign-up opens Thursday, due Sunday 5 PM
- A few of our SLs will be hosting a Qt Creator install help session tomorrow (Thursday) from 7-9 PM in Durand 353. See details in Jonathan's [Week #1 Announcements post](#) on Ed.
- Your first lecture quiz unlocked today, and it will be due next Wednesday at 1:00 PM. Please be sure to carefully track all deadlines this quarter. It's up to you to keep on top of all deadlines, even if there are not always reminders of those deadlines in class.

Productivity Tip: Hotkeys

I mentioned today that learning hotkeys for a few common tasks is critical for maximizing your coding productivity. Here are a few I brought up at various points in class today:

- **CTRL+R** will compile and run your project.
- **CTRL+I** will fix up the indentation of any code you have selected/highlighted.

Preliminaries: Syntax vs. Semantics

Before diving into C++ basics, I defined the following terms:

- **syntax:** the rules for constructing grammatical statements in a language (whether a coding language or a [natural language](#))
- **semantics:** meaning

Supplementary information:

Consider, for example, the following sentence:

The Count of Monte Cristo is on the shelf by *The Picture of Dorian Gray*.

There's a lot of stuff happening with that sentence, syntactically. The punctuation at the end of the sentence is part of the syntax, as is the italicization. The fact that we have a noun ("The Count of Monte Cristo"), followed by a verb ("is"), followed by a prepositional phrase ("on the shelf by 'The Picture of Dorian Gray'"), is also a matter of syntax. That is a grammatically sound sentence.

The semantics of that sentence relate to its meaning. We know, for example, that there isn't some old count (i.e., a person) sitting on a shelf next to a picture, where the picture happens to be of some dude named Dorian Gray. Semantically, what the sentence actually conveys is that there's a book on the shelf, and it's next to another book.

Why is this relevant? As I introduce you to different things you can do with the C++ programming language, I will generally introduce you to the *syntax* you have to follow (i.e., how to write lines of code that the compiler will be happy with) and the associated *semantics* (i.e., what that syntax *does* or *means*).

Preliminaries: Compilation and Execution

A **compiler** is a program that takes code and builds it into a program that we can run. When we hit CTRL+R in the Qt Creator, our code is first run through a compiler. If compilation is successful, the program then gets executed.

This is distinct from Python, where we often have an interpreter that executes our code line-by-line as we go.

Getting Started: The Blank Qt Project (and the return of "Hello, world!")

After some preliminary definitions, we dove into C++ code.

To kick that off, I went to the course homepage and downloaded the Blank Qt Project (also linked from the Resources page). This will be part of your normal process for creating new projects that use goodies from the Stanford C++ Library.

Next, I plunked down the following code from Monday:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

From there, we started exploring what was happening on each line of code.

Two Types of Comments in C++

I used comments extensively today to help communicate new concepts. We saw that C++ supports two commenting styles:

Firstly, we have what's called a **block comment**. If we type `/*`, then everything after that is considered a comment until we first encounter a `*/`. For example:

```

/* This is a comment!
 * This is also part of the same comment!
 * Qt Creator adds a star on each line for formatting purposes when we use this
 * type of comment.
Even though we don't have a star at the beginning of this line, it's still part of
our comment!
 * When we hit the end of this line, the star-slash causes our comment to end. */

#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}

```

Secondly, we have what's called a **single-line comment**. If we type `//`, then everything after that is considered a comment until we reach the end of that line. For example:

```

// This is a comment!
// This is a comment, too, but the line below this one is not!
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    cout << "Hello, world!" << endl; // This is a comment, too!
    return 0;
}

```

Glance over this! A Brief Guide to Commenting

We want you to comment your code throughout the quarter, but don't go overboard. Comments should be used to enhance the readability of your code. Avoid comments that point out fundamentals of the language that the grader will already know. For example:

Warning! All the comments below are unnecessary and should be removed. Please do not look to these as examples of useful comments!

```

// include statements
#include <iostream>
#include "console.h"
using namespace std;

// main function
int main()
{
    // prints "Hello, world!" to screen
    cout << "Hello, world!" << endl;
    return 0;
} // end of function

```

Below are some over-branching guidelines for writing helpful comments. Your SL will also give you feedback on comments when grading your assignments.

- In general, consider giving a high-level overview of any function you write whose purposes might not be immediately clear from its name. Say something about its input parameters, expected output, and return value (if applicable).
- Within your functions, rather than commenting every single line of code, give high-level comments that tell what *chunks* of code do. However:
 - Avoid simply repeating things that are "obvious" in code, thereby making someone read your code twice (once in English and once in C++). I.e., avoid [transliterating](#) your code from C++ to English.
 - Prefer comments that tell how/why something works rather than simply saying *what* something does, unless you're explaining the behavior of a chunk of code in a way that makes your code easier for a reader to digest.
- Try to use function names that are verb phrases, which therefore act as documentation of what your code is doing. (Avoid obscure function and variable names that would require us to add clarifying comments)

everywhere they appear.)

- Keep comments brief. An exception here is if you're writing a comment at the top of a sophisticated source file -- which is often a bit longer -- or documenting a particularly complex function.
- A good rule of thumb: Suppose you load up your code for an assignment five years from now, and you've lost the write-up for that assignment. What comments would help you quickly understand what you were doing with each of those functions?
- Another rule of thumb: If you find yourself really struggling to fix a bug in your code at some point, that *might* be a signal that you've found a complex bit of code that would be more clear if it had a concise comment.

#include Directives

In every language, we need to be able to import libraries of pre-written code. In Python and Java, we accomplish that with `import` statements. In C++, we use `#include` statements. The syntax for those is as follows:

including standard system libraries:

```
#include < LIBRARY_NAME >
```

including user-defined libraries:

```
#include " LIBRARY_NAME "
```

The Stanford C++ Libraries we `#include` this quarter will be in "double quotes," as they are user-defined libraries that we have piled on top of the standard goodies that you normally get with C++. The standard C++ libraries will be in <angled brackets>.

You needn't worry too much about this for now. In all your assignments, we will give you the `#include` statements you need so you don't have to hunt down the library for every single function you want to call.

(*Not mentioned in class.*) These `#include` directives are different from the other statements we saw today in that they are not terminated with semicolons. When it comes to `#include` statements, C++ always just reads those until it hits the end of the line -- no semicolon needed.

(*Not mentioned in class.*) I didn't mention this in class, but here's what we're using those particular libraries for:

- **<iostream>** gives us access to `cout` and `endl`
- **"console.h"** gives us a cute pop-up terminal for our program's output

Namespaces

(*Important Note*) You don't need to be super familiar with how namespaces work at this time. As long as you're comfortable writing `using namespace std;` in your programs, you're good to go. I'm including this section of notes because I never want to run the risk of leaving you frustrated, confused, or dissatisfied if you're the kind of person who loves to peek under the hood and really understand how everything is working. I personally find that being exposed to these sorts of details make me feel more comfortable using a new language.

When we work on large software projects, we often import multiple libraries from various sources. One of the problems that arises there is that some libraries might have functions with the same names as one another.

For example, a networking library that establishes connections to other computers on a network might have a `createConnection()` function, and a database library that establishes connections to databases stored on our machines might also have a `createConnection()` function. If we build a project that relies on both those libraries and then call `createConnection()`, we need a way to determine which version of that function gets called.

C++ uses namespaces to handle this issue. A namespace is just a named collection of functions (and other entities) that are grouped together. A networking library might articulate a `net` namespace, and a database library might articulate a `db` namespace. We would then call those `createConnection()` functions like so:

```
net::createConnection();  
db::createConnection();
```

The features of C++ we'll be using this quarter are pretty much all defined in a namespace called `std` (for "standard"). That includes features of C++'s standard libraries as well as the Stanford C++ Libraries. For example, `cout` and `endl` are both in the `std` namespace. To use them, we could do this:

```
#include <iostream>
#include "console.h"

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

The problem is, we'll be using a *lot* of functions and other goodies from those libraries, and having to type `std::` for each one of them will be a huge waste of time and lead to unnecessarily lengthy, complicated lines of code.

To get around that, we issue the `using namespace std;` statement above our function definitions. That tells C++ to look in the `std` namespace for the library components we use to build our programs, without the need to type `std::` in front of any of them. Here's the resulting change:

```
#include <iostream>
#include "console.h"

using namespace std;

int main()
{
    // removed both occurrences of std:: below
    cout << "Hello, world!" << endl;
    return 0;
}
```

If we get rid of both the `using namespace std;` statement and the `std::` tags, our program will not compile:

```
#include <iostream>
#include "console.h"

int main()
{
    // removed both occurrences of std:: below
    cout << "Hello, world!" << endl;
    return 0;
}
```

Because we've given no indication of what namespace to pull `cout` and `endl` from, the compiler gives us the following errors:

```
error: 'cout' was not declared in this scope; did you mean 'std::cout'?
error: 'endl' was not declared in this scope; did you mean 'std::endl'?
```

The `main()` Function

Next, we explored the definition of our program's `main()` function.

In the math world, a function does three main things:

1. It takes an input value (or, in some cases, multiple input values).
2. It does some work. For example, the function $f(x) = x^2$ multiplies x by itself (i.e., it squares x).
3. It spits out a result. For example, for $f(x)$ defined above, the result generated by $f(5)$ is 25.

Functions in C++ effectively do the same thing, but C++ doesn't like mystery or surprises when it comes to its functions' inputs and outputs. It wants us to say what *type* of parameter(s) we're going to pass to a function and what *type* of data a function will produce (called its **return type**).

The syntax for defining a function is as follows:

```
DATA_TYPE FUNCTION_NAME ( PARAMETERS )
{
    STATEMENT(S)
}
```

(*Key Take-Away*) Notice the need for opening and closing curly braces. In Python, we simply indent lines of code to signify that they make up the body of some function. In C++, we use a **code block** (a set of opening and closing curly braces that encapsulate lines of code).

(*Key Take-Away*) In C++, whitespace (in most cases) is not for the compiler; it's for humans. Please be kind to humans. Use good spacing.

(*Style Requirement*) Any time you open a new code block, the code within that block should be indented one level deeper than the braces themselves.

All this information offers us context for what's going on in `main()`. We return to the idea of writing new functions toward the end of today's notes, as well.

The `main()` Function is Special

The C++ compiler goes line-by-line through our code, starting on the first line and working its way down.

When we run a program that's written in C++, however, execution always begins in `main()`. That function gets executed automatically, without any need for us to call it manually. A program that doesn't have a `main()` function won't compile.

Return Statements

The syntax for returning a value from a function is as follows:

```
return EXPRESSION ;
```

The `EXPRESSION` can be any expression that evaluates to the appropriate return type for our function and can include, among other things, variables, arithmetic operations, hard-coded values ("literals"), and other function calls.

(*Key Take-Away*) Recall that `main()` should always return 0 (zero) upon successful execution of a program. You can think of this as `main()`'s way of saying that it encountered zero errors. That value is sent back to the operating system and can be monitored and interpreted by other programs.

We sometimes return non-zero values from `main()` if something goes awry and we want to signal to anyone listening that our program didn't accomplish what we wanted it to.

The `cout` Stream

`cout` is what we call a **stream**. By default, it acts as a direct line to our terminal. Anything we send to it with the `<<` operator gets printed to the screen. We saw today three primary things we can send to `cout`:

- string literals (a hard-coded string that appears in double quotes in our code, such as "Hello, world!")
- variables
- `endl`

Several examples are included throughout today's notes.

The `endl` String Manipulator

`endl` causes a newline character to print to the screen. Without it, all our output runs together. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    // Note the lack of endl in the following statements.
    cout << "Hello, world!";
    cout << "CS106B is awesome, and so are you!";

    return 0;
}
```


The output for that program is as follows:

```
Hello, world!CS106B is awesome, and so are you!
```

Here's the fix, with changes highlighted in orange (or is it peach?):

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    cout << "CS106B is awesome, and so are you!" << endl;

    return 0;
}
```

Our output now becomes:

```
Hello, world!
CS106B is awesome, and so are you!
```

Variables and Data Types

Unlike Python, which is very flexible with variable usage, C++ requires us to formally declare every variable we want to use and give it a specific data type that cannot be changed after declaration. We saw today the following data types:

data type keyword	type description	example(s)
int	integer (whole number)	-3, 0, 1, 1051
float	floating point number (real number)	3.14159
double	floating point number (real number)	3.14159
char	character (single glyph)	'q', 'P', '\$' (requires single quotes)
string	string of characters	"Hello, world!" (requires double quotes)
bool	Boolean	true, false (no quotes)

A **variable declaration** is a statement wherein we bring a new variable into existence and give it a data type. The syntax for that in C++ is:

```
DATA_TYPE  VARIABLE_NAME ;
```

See examples below.

Notice that we can mix string literals with variables in our *cout* statements, and we can output multiple variables with a single *cout* statement, as well:


```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int a = 5;
    char ch = 'q';
    float f = 3.14159;
    double d = 3.14159;
    string s = "hello";

    cout << "Hello, world!" << " " << a << endl;
    cout << ch << " " << f << " " << d << endl;
    cout << s << endl;

    return 0;
}
```

Side note: A `double` typically uses double the amount of memory that a `float` uses and can therefore store more digits after the decimal place. You do not need to know that for this class. That topic is explored further in CS107.

Common Pitfalls with Variables

Here are some common errors we see with variables:

Undeclared variable. We cannot use a variable that has not been declared. This code will not compile.

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    a = 5; // ERROR: a is undeclared; we must give it a data type

    cout << a << endl;

    return 0;
}
```

Type mismatch. In this example, `a` was designed to hold an integer. We cannot set it equal to a string.

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int a = 5;
    a = "hello"; // ERROR: type mismatch; a can hold an integer, not a string

    cout << a << endl;

    return 0;
}
```

Attempting to change a variable's data type. Once a variable is declared to be of a certain type, it cannot be redeclared with a different type.

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int a = 5;
    string a = "hello"; // ERROR: a is already declared; cannot declare with new type

    cout << a << endl;

    return 0;
}
```

Redefinition of variables. Here, we are trying to create a twice.

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int a = 5;
    int a = 7;  // ERROR: a is already declared on the line above

    cout << a << endl;

    return 0;
}
```

If you want to change the value of a variable you've already declared, simply do it on another line without re-declaring the data type, like so:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int a = 5;
    a = 7;  // OKAY

    cout << a << endl;

    return 0;
}
```

Uninitialized Variables

In C++, variables are not auto-initialized for you. By default, if you don't assign a value to a variable, it contains garbage. (An exception here is string variables, which are empty by default.)

Most C++ compilers will merrily allow you to compile code in which variables have not been initialized. The Qt Creator will not -- and so I (as someone who doesn't typically use an IDE when writing C++ code) was caught off guard the first time I went to compile the following in the Qt Creator, and it actually stopped me:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int a;
    int b;
    int c;

    cout << a << " " << b << " " << c << endl;  // YIKES. Uninitialized variables.

    // Some compilers will let this compile despite the uninitialized variables!

    return 0;
}
```

While Loops

We assume that everyone coming into this course has seen while loops before, and so I introduced the syntax rather quickly.

Syntax:

```
while ( CONDITION )
{
    STATEMENT(S)
}
```

Note that the parentheses around the condition are required.

Example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int i = 1;

    while (i < 5)
    {
        cout << i << endl;
        i++;
    }

    return 0;
}
```

In Python, we would have done something like this:

```
// Python code! Not C++!
i = 1
while i < 5:
    print(i)
    i += 1
```

The ++ Operator

Notice above that we used `i++` instead of `i += 1`. It is so common to want to increment an integer variable by one that C++ has a dedicated operator (the `++` operator) to do just that.

Wonky Spacing

To reinforce that C++ doesn't worry too much about whitespace, I at one point showed that a program still compiles even if we do something totally wonky with our spacing, like this:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int i = 1;

    while (i < 5)
    {
cout << i << endl;

        i++;
    }

    return 0;
}
```

We also saw that if we highlight a chunk of code with wonky spacing, the Qt Creator is able to fix the indentation for us. Hit `CTRL+I` to fix indentation (or, if you can't remember the shortcut, right click and find that option in the menu that pops up).

(*Not covered in class.*) Similarly, although I did not cover this in class, C++ would compile the following:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int i = 1;
    while (i < 5) { cout << i << endl; i++; }
    return 0;
}
```

(*Style Requirement*) The above approach is difficult to read, and putting multiple statements on a single line is actually

prohibited in our style requirements for programming assignments.

For Loops

C++ also has for-loops, although they are quite different from Python's for-loops. Here's the syntax:

```
for ( INITIALIZATION_STATEMENT ; CONDITION ; POST_ITERATION_STATEMENT )  
{  
    STATEMENT(S)  
}
```

Notice the need for semicolons between each of the three components of the for-loop. The initialization statement is executed exactly once (when we first hit the for-loop), the condition is evaluated before kicking off an iteration, and the post-incrementation statement is executed at the end of each iteration of the loop, right before re-evaluating the looping condition.

Example:

```
#include <iostream>  
#include "console.h"  
using namespace std;  
  
int main()  
{  
    for (int i = 1; i < 5; i++)  
    {  
        cout << i << endl;  
    }  
  
    return 0;  
}
```

This for-loop is *syntactically* distinct from the while-loop above, but it is *semantically* the same (aside from the fact that the `i` variable is out of scope once we leave the for-loop, which we discussed at the very end of lecture).

For-Each Loops (Range-Based Loops)

This is a third type of loop that C++ offers. It's good for looping through all the elements in a container (such as all the characters in a string or all the elements in an array). We will see additional containers this quarter that are amenable to this sort of loop.

Syntax:

```
for ( DATA_TYPE VARIABLE_NAME : CONTAINER )  
{  
    STATEMENT(S)  
}
```

Example:

```
#include <iostream>  
#include "console.h"  
using namespace std;  
  
int main()  
{  
    string s = "giraffe";  
  
    // Initially, ch is set equal to the first character in s. With each subsequent  
    // iteration of the loop, ch moves forward by one character in the string.  
    for (char ch : s)  
    {  
        cout << ch << endl;  
    }  
  
    return 0;  
}
```

This produced the following output, where the string is tall and elongated, just like a giraffe. :)

```
g
i
r
a
f
f
e
```

Conditional (If-Else) Statements and Comparison Operators

The syntax for an if-statement in C++ is very similar to that of a while-loop.

Note that the parentheses around the condition are required.

Syntax:

```
if ( CONDITION )
{
    STATEMENT(S)
}
```

In class, we started with a small example and continued to extend it until we reached the following:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int numCupcakes = 5;

    if (numCupcakes == 1)
    {
        cout << "Oh no! Running low!" << endl;
    }
    else if (numCupcakes > 1)
    {
        cout << "Hooray, cupcakes!" << endl;
    }
    else
    {
        cout << "Oh noo0o0o00oo!" << endl;
    }

    return 0;
}
```

The comparison operators in C++ are:

operator	meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Note that the <= operator cannot be written as =< (with the equal sign up front). The same holds for the >= operator. One way to remember this (mnemonic time!) is that it's far more common to say "less than or equal to" in English than "equal to or less than." The order of the operators in C++ follows the conventional order of the adjectival phrases in English.

Playing with each of these is left as an exercise.

Boolean Operators

At the very end of class today, we briefly examined the following three Boolean operators.

operator	meaning
!	Boolean "not"
&&	Boolean "and"
	Boolean "or"

The `!` operator (the "not" operator) inverts a boolean value; `true` becomes `false`, and `false` becomes `true`. Accordingly, the following chunk of code:

```
if (numCupcakes != 13)
{
    cout << "Not a baker's dozen." << endl;
}
```

... is semantically equivalent to:

```
if (!(numCupcakes == 13))
{
    cout << "Not a baker's dozen." << endl;
}
```

Notice that when applying the `!` operator to a whole expression (rather than a single variable), we use parentheses to wrap up the entire expression we wish to negate.

The `&&` operator (the "and" operator) evaluates to `true` if, and only if, both of its operands are `true`. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int numCupcakes = 13;
    bool stillMakingPoorLifeChoices = true; // om nom nom

    if (numCupcakes == 13 && stillMakingPoorLifeChoices)
    {
        cout << "Hooray, cupcakes!" << endl;
    }

    return 0;
}
```

The `||` operator (the "or" operator) is `true` if at least one of its operands is `true`. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int numCupcakes = 13;

    if (numCupcakes == 1 || numCupcakes == 2)
    {
        cout << "Uh oh! We're running low on cupcakes!" << endl;
    }

    return 0;
}
```

(*Important Note!*) You must use two ampersands for the `&&` operator and two pipes for the `||` operator. If you just use a single `&` or `|`, you actually get a different operation. (The `&` and `|` operators are actually bitwise operators, which are discussed in CS107.)

Common Pitfall with the `||` Operator

In the following example, the `cout` statement is executed because C++ considers the value `2` to be `true`. If we want to check whether a variable is equal to one value or another, we need to apply the `==` operator twice.

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int numCupcakes = 5;

    if (numCupcakes == 1 || 2) // Yikes! This evaluates to true!
    {
        cout << "Uh oh! We're running low on cupcakes!" << endl;
    }

    return 0;
}
```

Here's the correct version:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int numCupcakes = 5;

    if (numCupcakes == 1 || numCupcakes == 2) // Fixed!
    {
        cout << "Uh oh! We're running low on cupcakes!" << endl;
    }

    return 0;
}
```

Catch-up Topic Void Functions

In this section, we define a few functions other than `main()` and explore return types, return values, and parameters.

If a function takes no parameters, we simply leave the parameter list within its parentheses blank.

If we write a function that does some work for us, but which we never expect to produce a meaningful value at the end of its execution, then it needn't return a value at all. In this case, we give the function a return type of `void`, like so:

```
#include <iostream>
#include "console.h"
using namespace std;

void greet()
{
    cout << "hello :)" << endl;
}

int main()
{
    greet();
    return 0;
}
```

A `void` function does not require a `return` statement. We return to the function that called it when we reach the end of the function's definition.

(*Not covered in lecture.*) Optionally, if we wish to leave a `void` function before its final line, we can simply return (without specifying a return value), like so:


```
#include <iostream>
#include "console.h"
using namespace std;

void processCupcakes(int numCupcakes)
{
    if (numCupcakes < 0)
    {
        cout << "Invalid number of cupcakes." << endl;
        return;
    }

    // One can imagine doing something useful with numCupcakes here.
    // If we pass a negative value to this function, the following line
    // will not print.
    cout << "We reached the last line of the function." << endl;
}

int main()
{
    processCupcakes(-3);
    return 0;
}
```

Catch-up Topic Passing Parameters and Returning Values from Functions

The following `square()` function is designed to take a single integer parameter, `x`, and return x^2 . To set this function up to accept a parameter, we give a full declaration for that variable within the parentheses in the function signature (the line with the function's name, return type, and parameter list).

```
#include <iostream>
#include "console.h"
using namespace std;

// We pass an integer to this function, and it returns an integer.
int square(int x)
{
    return x * x;
}

int main()
{
    square(5);
    return 0;
}
```

However, when we run the above program, nothing gets printed to the screen. That's because we never did anything in `main()` with the return value of our call to `square(5)`. We must capture that return value. Here are two ways to print the return value to the screen:

```
#include <iostream>
#include "console.h"
using namespace std;

// We pass an integer to this function, and it returns an integer.
int square(int x)
{
    return x * x;
}

int main()
{
    // Option 1: Send the return value of square(5) directly to cout.
    cout << square(5) << endl;

    // Option 2: Store the return value in a variable and print that.
    int result = square(5);
    cout << result << endl;

    return 0;
}
```

The program now produces the following output:

```
25
25
```

(*Not covered in lecture.*) If we want our function to print the result directly, we could write it as follows. Notice that the modified behavior of this function is reflected both in its modified name and the `void` return type:

```
#include <iostream>
#include "console.h"
using namespace std;

void printSquare(int x)
{
    cout << x * x << endl;
}

int main()
{
    printSquare(5);
    return 0;
}
```

Catch-up Topic Function Placement and Functional Prototypes

I mentioned toward the beginning of class that the C++ compiler processes your code line-by-line, starting at the very first line. A consequence of that is that if the compiler encounters a function call for a function that hasn't been defined yet (or for which we have not yet written the requisite `#include` statement), it goes kaput. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    cout << square(5) << endl; // ERROR: square() function not yet defined.
    return 0;
}

int square(int x)
{
    return x * x;
}
```

What's happening above is that when we hit the `cout` line, C++ goes, "What's this `square()` function? Never heard of it." And it stops trying to compile the program in protest.

There are two ways around this:

1. Move the definition of the `square()` function above `main()` .
2. Introduce a functional prototype!

A **functional prototype** is just a function signature with a semicolon. A **function signature** is that first line of your function's definition that gives the return type, function name, and parameter list. Here's a functional prototype in action:

```
#include <iostream>
#include "console.h"

using namespace std;

// functional prototype
int square(int x);

int main()
{
    cout << square(5) << endl; // This is okay now.
    return 0;
}

int square(int x)
{
    return x * x;
}
```

This time, when C++ reaches the `cout` line, even though it doesn't have a definition for the `square()` function, it has a good idea (from having already seen the functional prototype) of how that function should be called, including the number and type of arguments it takes and its return type. It checks out the `cout` line and says, "Alrighty. I don't know how the `square()` function works yet, but I do know that the way you're calling it here fits with how I expect it to be called. Good work." It then carries on and successfully compiles the program.

(*Not covered in class.*) Note that if we never defined the `square()` function, the program would not compile, even if we had a functional prototype. The compiler would fail at the very end of the file, when it realized there was no

more code to process and it never got a definition for `square()` :

```
#include <iostream>
#include "console.h"

using namespace std;

// functional prototype
int square(int x);

int main()
{
    cout << square(5) << endl;
    return 0;
}

// ERROR: Reached end of code without ever defining square() function.
```

Catch-up Topic Variable Scope

Variables only exist within the code blocks where they are declared -- or, in the event that you declare a variable in the header of a for-loop, that variable only exists within the loop. We refer to the regions of code that can refer to a particular variable as the variable's **scope**.

For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    for (int i = 0; i < 5; i++)
    {
        int q = 5;
        cout << i << endl;
    }

    cout << "i is: " << i << endl; // ERROR: undeclared variable
    cout << "q is: " << q << endl; // ERROR: undeclared variable

    return 0;
}
```

In the code above, we say that the `i` and `q` variables only exist within the scope of the for-loop. To refer to them outside the for-loop, they would have to be declared elsewhere. For example:

```
#include <iostream>
#include "console.h"
using namespace std;

int main()
{
    int i;
    int q;

    for (i = 0; i < 5; i++)
    {
        q = 5;
        cout << i << endl;
    }


    cout << "i is: " << i << endl; // OKAY
    cout << "q is: " << q << endl; // OKAY

    return 0;
}
```

What's next?

On Friday, we continue our fast-paced introduction to C++! We'll talk about the "catch-up" topics indicated above, as well as pass-by-value functions, pass-by-reference, strings in C++, and some things related to testing our code.

Exam Prep

1. After reviewing today's notes, open a copy of the  [blank Qt Creator project](#), delete all the lines of code it contains, and try to write a "Hello, world!" program from scratch. Try to do this without peeking at your notes. After that, be sure to spend some time coding up your own examples from scratch. Play with the various operators and data types we saw in class today, and be sure to write loops and conditional statements to solidify your understanding of the syntax for those things in C++.
2. Be sure to glance through the notes labeled "*(Not covered in class.)*" and **Glance over this!** to enrich and/or solidify your understanding of some of the finer points of this material. Those extra nuggets of information might prove useful at some point this quarter.
3. Be sure to glance through the notes labeled **Catch-up Topic** . Those give a sneak preview of the first few topics we will cover in our next lecture.

All course materials © Stanford University 2024. This content is protected and may not be shared, uploaded, or distributed.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-02